Travaux Pratiques
Programmation Multi-Paradigme
Licence 3 Informatique

Julien BERNARD

Table des matières

Consignes communes à tous les projets

Au cours de cette UE, vous avez **trois** projets à réaliser à raison d'un projet pour deux séances de trois heures de travaux pratiques encadrées. Les projets sont à faire et à rendre dans l'ordre du présent sujet.

Pour chaque projet, vous devrez implémenter une interface donnée dans un fichier d'en-tête, ainsi qu'un ensemble de tests unitaires pour cette interface. Les tests serviront à montrer que votre implémentation est correcte et complète.

Projet n°1 : *Smart Pointers*

En C++, lorsqu'on alloue dynamiquement un objet, l'utilisateur doit le libérer quand il n'est plus utile. Il ne faut pas le libérer trop tôt afin d'éviter des accès invalides (souvent traduit par des *segfaults*). Il ne faut pas non plus trop attendre sous peine de saturer la mémoire ; dans le pire des cas, l'objet n'est jamais libéré ce qui entraîne des fuites mémoires.

Avec l'arrivée du C++ moderne, on a mis au point des pointeurs intelligents *(Smart Pointers)*. Ce concept se base le notion de propriété *(ownership)* d'un objet dynamique. Un pointeur intelligent possède un objet dynamique lorsqu'il est responsable de l'accès et de la libération de l'objet.

Le but de ce premier projet est d'implémenter trois types de pointeur intelligent :

1. `Unique`
2. `Shared`
3. `Weak`

Étape 1 : *Unique pointer*

Le premier type de pointeur intelligent que nous allons implémenter est le pointeur `Unique`. Lorsqu'un pointeur `Unique` possède un objet dynamique, aucun autre `Unique` pointeur ne peut le possèder. Cela signifie qu'à tout instant du programme, un objet dynamique ne peut être possédé que par un seul pointeur `Unique`.

Par conséquent, on ne peut pas copier un pointeur `Unique` car cela signifierait que deux pointeurs `Unique` possèdent le même objet. Toutefois, il est possible de déplacer un pointeur `Unique` vers un autre. Dans ce cas, la propriété de l'objet est transférée entre les pointeurs `Unique`. Lorsqu'un pointeur `Unique` est détruit, l'objet qu'il possédait est libéré.

Pour plus de détails sur le fonctionnement d'un pointeur `Unique`, vous pouvez aller vois la classe équivalente de la bibliothèque standard : `std::unique_ptr`.

Étape 2 : *Shared pointer*

Le deuxième type de pointeur que nous allons voir est le pointeur `Shared`. Plusieurs pointeurs `Shared` peuvent posséder un même objet dynamique. Chacun des ces pointeurs peut accéder et modifier l'objet dynamique.

Il est donc tout à fait possible de copier un pointeur `Shared` créant ainsi un pointeur `Shared` gérant le même objet dynamique. On peut également déplacer un pointeur `Shared` et de ce cas là, le pointeur source transfert la possession de l'objet au nouveau pointeur. L'objet dynamique n'est libéré que lorsque le dernier pointeur `Shared` le possédant est détruit. Tant qu'il reste au moins un pointeur `Shared` qui possède l'objet, il est gardé en mémoire.

Pour cela, il est nécessaire d'avoir un compteur du nombre de pointeurs `Shared` qui pointent vers le même objet. Le compteur accompagne le pointeur alloué. Quand ce compteur tombe à zéro, on peut libérer l'objet.

Pour plus de détails sur le fonctionnement d'un pointeur `Shared`, vous pouvez aller vois la classe équivalente de la bibliothèque standard : `std::shared_ptr`.

Étape 3 : *Weak pointer*

Le dernier type de pointeur que nous allons ajouter sont les pointeurs `Weak`. Ce type de pointeur fonctionne de pair avec les pointeurs `Shared`. En effet, un pointeur `Weak` ne possède pas réellement l'objet alloué, il doit s'assurer que l'objet existe encore avant de pouvoir y accéder. S'il existe encore, il générera un nouveau pointeur `Shared` possédant l'objet dynamique.

L'intérêt des pointeurs `Weak` vient du fait qu'un cycle de pointeurs `Shared` ne peut pas être libéré. L'introduction d'un pointeur `Weak` permet de casser le cycle et donc de permettre la libération de tous les pointeurs.

On peut copier et déplacer les pointeurs `Weak` comme on l'a fait pour les pointeurs `Shared`. En revanche, l'objet dynamique est libéré lorsque le dernier pointeur `Shared` qui le possède est détruit. Il est donc tout a fait possible d'avoir des pointeurs `Weak` qui font référence à un objet libéré. C'est pourquoi il est impératif de vérifier l'existence de l'objet avant d'y accèder. À l'inverse, il est possible de détruire tous les pointers `Weak` qui font référence à un objet dynamique sans que celui-ci ne soit libéré.

En pratique, il faut, en plus du compteur d'objets précédemment décrit, ajouter un compteur de pointeurs `Weak` qui servira à savoir quand libérer les compteurs associés à un objet.

Pour plus de détails sur le fonctionnement d'un pointeur `Weak`, vous pouvez aller vois la classe équivalente de la bibliothèque standard : `std::weak_ptr`.

Exemple d'utilisation

```cpp
#include <iostream>

#include "Shared.h"
#include "Unique.h"
#include "Weak.h"

int main() {
  sp::Unique<int> unique(new int(0));

  ++(*unique);
  std::cout << *unique << std::endl; // 1

  sp::Shared<int> shared(new int(42));
  std::cout << *shared << std::endl; // 42

  sp::Weak<int> weak1(shared);
  {
    auto tmp = weak1.lock();
    bool b = tmp.exists(); // true
    (*tmp) /= 2;
    std::cout << *tmp << std::endl; // 21
  }

  shared = sp::Shared<int>(new int(1337));
  sp::Weak<int> weak2(shared);
  {
    auto tmp = weak1.lock();
```

```
    bool b = tmp.exists(); // false

    tmp = weak2.lock();
    std::cout << *tmp << std::endl; // 1337
  }

  return 0;
}
```